

Parallel Galton Watson Process

Olivier Bodini, Camille Coti, and Julien David

LIPN, CNRS UMR 7030, Université Paris 13,

Sorbonne Paris Cité

Villetaneuse, France

\protect\T1\textbraceleftfirstname.lastname\protect\T1\textbraceright@lipn.univ-paris13.fr

Abstract—In this paper, we study a parallel version of Galton-Watson processes for the random generation of tree-shaped structures. Random trees are useful in many situations (testing, binary search, simulation of physics phenomena,...) as attests more than 49000 citations on Google scholar. Using standard analytic combinatorics, we first give a theoretical, average-case study of the random process in order to evaluate how parallelism can be extracted from this process, and we deduce a parallel generation algorithm. Then we present how it can be implemented in a task-based parallel paradigm for shared memory (here, Intel Cilk). This implementation faces several challenges, among which efficient, thread-safe random bit generation, memory management and algorithmic modifications for small-grain parallelism. Finally, we evaluate the performance of our implementation and the impact of different choices and parameters. We obtain a significant efficiency improvement for the generation of big trees. We also conduct empirical and theoretical studies of the average behaviour of our algorithm.

I. INTRODUCTION

Branching processes are very simple and natural procedures that models evolution of individuals. Such a process is extremely popular and emerges in a lot of situations; for example, in epidemiology, where individuals correspond to bacteria or in genealogy, (the initial study of Galton-Watson was the spread of surnames), but also in physics, to simulate the propagation of neutrons during a nuclear fission.

In this paper, we focus on two standard branching processes. Firstly, on the most common Galton-Watson process which corresponds to halt with probability $\frac{1}{2}$ or generation of 2 sons with probability $\frac{1}{2}$. This process appears in computer science as a good model of random rooted planar binary trees, indeed the tree obtained by writing the genealogy of the offspring conditioned to have a fixed number n of individual is known to be uniform over the set of all binary trees of size n .

Secondly on the process that generates uniformly rooted planar binary increasing trees. This choice is motivated by the fact that increasing trees are central data structures and arise in a huge number of important algorithms such as binary search algorithms, image segmentation, natural language processing, ...

It is therefore of crucial interest to be able to sample very large branching processes efficiently. For instance, nuclear simulations need very huge sampling. Now, let us observe that this process are intrinsically parallel, due to independence of each branch. But quite surprisingly, this paper is the first attempt to describe a parallel version and to the best of our

knowledge, there are currently few research results on this subject [3] (note that we do not include random numbers as combinatorial objects). For the sequential version, let us notice that the problem can be reduced to Boltzmann sampling [5], [2] and in the size-conditioned case has been tackled by [10], [1] for binary trees and [9] for increasing binary trees. In this paper, we give and study the first parallel algorithm that produces binary trees and increasing binary trees with a quasi-perfect distribution of load over the cores of a multi-core processor. Up to our knowledge, this paper is the first example of analyzing in distribution in the domain of parallel algorithms.

Although Galton-Watson processes seem to be easy to parallelize, in practice a parallel implementation is really not trivial. As a matter of fact, average-case analysis results presented in this paper show that parallelism is fine-grained: the average work done by each thread is constant (or logarithmic in the increasing case) and parametrized by a threshold. This threshold characterizes a time when it is necessary that some works are given to another thread.

This paper is organized in three main parts. Section II describes an algorithm that parallelizes the Galton Watson Processes, with some small but important implementation details. Section III contains a short analysis of the main parameters of this algorithm, that is the lifetime of the threads, the peak load and the total time in fully parallel model. This part deals with conventional analytic combinatorics. We show that, considering an ideal context in which \sqrt{n} threads can run in parallel, our algorithm can sample a n -node Galton Watson tree in $\Theta(\sqrt{n})$ average time complexity. In section IV we present implementation details that helped us making efficient such a fine-grain parallelism.

II. ALGORITHMS

The classical and naive implementation of a Galton Watson process can be found in Algorithm 1. Though this method and its parallel version are not efficient, the author thought it would improve readability to recall its description in order to compare this version to the improved ones. When processing a node, the algorithm generates a random bit. If it is equal to 0, the node is a leaf. If it is equal to 1, the node is internal and has two subtrees.

As one can see, it is as simple to implement as it is inefficient. Double recursivity ensures to obtain a lot of movements on the call stack. Also, since Galton-Watson processes are

Algorithm 1: NaiveGaltonWatson

```

1 Data: a node  $n$ 
2 Result: A binary tree enrooted in  $n$ 
3  $b \leftarrow$  draw a random bit;
4 if  $b = 1$  then
5   | Add nodes  $n_1$  and  $n_2$  as node  $n$ 's children;
6   |  $\text{NaiveGaltonWatson}(n_1)$ ;
7   |  $\text{NaiveGaltonWatson}(n_2)$ ;
8 end if

```

branching ones, a natural way to obtain a parallel algorithm is the following: when a new node is created, its two subtrees are managed by two different threads (the main one and a new one). This method seems to be inefficient: as we will prove later in section III, the new thread immediately stops with probability $\frac{1}{2}$ (the subtree is a leaf), and produces a small subtree of size 4 in average.

A proper sequential implementation requires an iterative version of this algorithm 2, using a stack of nodes instead of the call stack. As we will see in benchmark section, this already drastically improves execution time.

Algorithm 2: IterativeGaltonWatson

```

1 Result: A binary tree
2  $lds_1 \leftarrow$  Create a linear data structure;
3 Create a tree with root  $r$ ;
4 Push  $r$  into  $lds_1$ ;
5 while  $lds_1$  is not empty do
6   | node  $n \leftarrow pop(lds_1)$ ;
7   |  $b \leftarrow$  draw a random bit;
8   | if  $b = 1$  then
9     | Add nodes  $n_1$  and  $n_2$  as node  $n$ 's children;
10    | Push  $n_1$  and  $n_2$  into  $lds_1$ ;
11    end if
12 end while

```

The idea of our algorithm is the following: parallel computation can improve efficiency but some of its aspects might have an overhead on the computation time. In a multi-threaded environment, waking up a sleeping thread or creating a new thread to perform part of the computation can indeed cost some time. Thus, one needs to make sure that the new called thread will not halt too quickly.

In order to compute in parallel efficiently, a thread should be spawned only if it has “enough” work to do, in a sense that we want to improve the average size of the generated subtree which is handled by a thread. Our idea is to use a data structure to accumulate nodes to process and spawn a new thread when it reaches a sufficient size, meaning that enough nodes are to be processed by this thread.

In Algorithm 3, we use two linear data structures lds_1 and lds_2 to keep track of the nodes that have to be processed. There is two advantages in doing so. First it allows us to obtain a version of the algorithm which is theoretically iterative inside a thread, which is faster than a recursive version. Then, it allows us to solve the aforementioned problem. New nodes to process

are pushed in lds_1 , until its size reaches a given threshold t . New nodes are then pushed in lds_2 . When it reaches size t , a new thread is spawned. This thread will manage the nodes gathered in lds_2 .

Algorithm 3: ParallelGaltonWatson (first called with a linear data structure containing the tree's root)

```

1 Data: a linear data structure  $lds_1$ , a threshold  $t$ 
2 Result: A binary tree
3  $lds_2 \leftarrow$  Create a linear data structure;
4 while  $lds_1$  is not empty do
5   | if  $lds_2$  is empty then
6     | node  $n \leftarrow pop(lds_1)$ ;
7     end if
8     else
9       | node  $n \leftarrow pop(lds_2)$ ;
10    end if
11     $b \leftarrow$  draw a random bit;
12    if  $b = 1$  then
13      | Add nodes  $n_1$  and  $n_2$  as node  $n$ 's children;
14      | if  $|lds_1| < t$  then
15        | Push  $n_1$  and  $n_2$  into  $lds_1$ ;
16        end if
17      else
18        | Push  $n_1$  and  $n_2$  into  $lds_2$ ;
19        | if  $|lds_2| \geq t$  then
20          | Start  $\text{ParallelGaltonWatson}(lds_2, t)$  on a
21          | new thread;
22          |  $lds_2 \leftarrow$  Create a new linear data structure;
23          end if
24        end if
25    end if
26 end while

```

Note that the size of lds_1 is at most increased by 1 at each iteration of the while loop. Therefore, when the algorithm starts filling lds_2 , we have $|lds_1| = t$.

A. Hybrid Algorithm

Algorithm 3 is slower than Algorithm 2 when small objects are generated. Indeed, the parallel version requires additional data structures whose cost is not negligible if the objects are small. Therefore, we decided to add an algorithm which is a merge of Algorithm 3 and Algorithm 2: at first the tree is generated sequentially, and once the linear data structure reaches a given size the program switches to the parallel implementation.

III. COMPLEXITY ANALYSIS

In this section, we deal with basic analytic combinatorics [8] in order to produce some results on the behavior of this algorithm. The first parameter that we would like to analyze is the peak total load of the processors, that is to say the maximum number of nodes in current treatment in all the threads. This analysis uses standard results on the maximum height of a Dyck path. We can easily deduce from this result that the peak load for the sampling of a tree of size n is in average in $O(\sqrt{n})$. We give more details later in this section. The second one is the time of the process assuming that we

have a massive parallel computation. That is to say that all thread operates in parallel from all the others. In this model, we prove in the sequel that our algorithm runs in average in $O(\sqrt{n})$. The third significant parameter is the average lifetime of the first thread. We restrict here our attention to the case where the threshold is equal to 1, 2 or 4, the complete analysis being laborious and out of the scope of this introducing paper. The first thread has the property to have in average the largest lifetime. It is a natural mean upper bound for all the other threads. Moreover, as we can see in the sequel, its mean lifetime is asymptotically constant, and consequently the mean lifetime is asymptotically the mean lifetime of almost all the threads. This ensures a good distribution of the load.

A. Peak total load

First, we want to recall the studied model. We begin with a queue containing one node, this node leave the queue and with probability 1/2 generate zero or two new nodes. We want to analyze the evolution of this linear data structure given the fact that we know that the process stops (the queue becomes empty) after having generated n nodes. This type of question arises in numerous situations, from statistical physics to urn process. It is a well known and classically called *one-dimensional Brownian excursion*. In particular, the peak load P is the maximum height of this excursion. It follows a Theta distribution (see [8] pp328, for definitions and details):

Theorem 1. *The peak load P of our algorithm follows after normalization by $\frac{1}{2\sqrt{\pi n}}$ a Theta law with expectation $\sqrt{\pi n}$.*

B. Time complexity in fully parallel model.

We just analyze two cases depending on the threshold is one or two. We have two reasons for this restriction. Firstly, experimentally, these both cases are the most convenient. Secondly, up to two, the analysis is much more tricky and cannot include in this conference version. Thirdly, due to universality of the parameters, we cannot expect great changes of behavior between one and two and the next values.

So, for threshold one and two, the time complexity problem reduces to very standard question. Indeed, in case of threshold one, it is a simple observation, that every node at level h (by convention, the root is at level 0) is treated after h operations. Thus, the time complexity is just the height of the generated tree. This very standard problem has been tackled by De Bruijn, Knuth, Rice [4], [6]. We come back to the same distribution that for peak load:

Theorem 2. *The time complexity T of our algorithm with threshold 1 in a fully parallel model follows after normalization by $\frac{1}{2\sqrt{\pi n}}$ a Theta law with expectation $\sqrt{\pi n}$.*

The next result about threshold 2 is just a remark. Indeed, every node at level h is treated after $2h - 1$ or $2h$ operations. This implies:

Theorem 3. *The time complexity T of our algorithm with*

threshold 2 in a fully parallel model follows after normalization by $\frac{1}{\sqrt{\pi n}}$ a Theta law with expectation $\sqrt{\pi n}$.

Notice that the algorithm with threshold 2 is theoretically slower than this with threshold. Nevertheless, our theoretical model does not take into account the fact that spawning a thread has a non negligible cost.

C. Lifetime of the first thread in the case of threshold 1 and 2.

1) *Threshold 1.*: We are going to mark the nodes which are treated by the first thread. Using standard approach by symbolic methods as presented in [8], we get the following specification for the marked class of tree:

$$\mathcal{T}^u = \mathcal{U}\mathcal{Z} + \mathcal{Z}\mathcal{U}\mathcal{T}^u\mathcal{T} \text{ and } \mathcal{T} = \mathcal{Z} + \mathcal{Z}\mathcal{T}^2$$

In other words, the thread processes nodes on the most left branch of the tree. The length of the left branch of a random tree is a classical problem in combinatorics. We give here a very fine analysis.

Let $T_{z,u} = \sum_{n,k} t_{n,k} z^n u^k$ be the bivariate generating function such that $t_{n,k}$ counts the number of trees of size n having a first thread of size k . By classical dictionary from specifications to generating functions, we know that $T_{z,u}$ is given by the functional equation:

$$T_{z,u} = uz + zuT_{z,u}T(z).$$

But $T(z)$ is nothing but the generation function of rooted binary trees, so $T(z) = \frac{1-\sqrt{1-4z^2}}{2z}$. We directly deduce that $T_{z,u} = \frac{2uz}{2-u+u\sqrt{1-4z^2}}$.

Using guess and prove strategy, we can easily derive that when n is odd $t_{n,k} = \frac{2(k-1)}{n-1} \binom{n-k-1}{\frac{n-3}{2}}$ and $t_{n,k} = 0$ otherwise. This strategy proceeds as follows: firstly, calculate the first values of $t_{n,k}$, factorize them, and observe that no large divisors appear. Generally, this means that the values are product of factorials. Quite easily, we find that it should be of the shape $\frac{2(k-1)}{n-1} \binom{n-k-1}{\frac{n-3}{2}}$. Secondly, we just have to prove that our guessing is correct. For this, we extract a system of linear recurrences that allowed to build the $t_{n,k}$ from the function equation $T_{z,u} = uz + zuT_{z,u}T(z)$. Indeed, this equation is algebraic, so it is also holonomic and verifies the differential equations:

$$\begin{cases} (u^2 z^2 - 8 z^2 u + 8 z^2 + 3 u - 3) t(z, u) + (4 u^2 z^6 - u^2 z^4 - 4 u z^4 + 4 z^4) t'(z, u) \\ + (2 * u^2 * z^2 - 4 * u * z^2) * (diff(t(z, u), u)) + (4 * z^3 - z) t''(z, u) = 0 \end{cases}$$

and the coefficients follows a P-recurrence.

We also extract a linear system of recurrences from the binomial expression:

$$\begin{cases} (k^2 - km - k) t_{2m+1,k} = (k^2 - 2km - k + 2m) t_{2m+1,k+1} \\ (k^2 - 4km + 4m^2 - 3k + 6m + 2) t_{2m+1,k} = (m^2 - km - k + 3m) t_{2m+1,k-1} \end{cases}$$

The last step is just to show that the two recurrences are equivalent.

Now, to reach the mean lifetime of the first thread for a tree of size n , it suffices to observe that it corresponds to the

$$\text{value } M_n = \frac{[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1}}{[z^n] T(z)} \quad 1. \text{ Indeed, } [z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1} = \sum_{n,k} k t_{n,k} z^n u^k. \text{ We have } \frac{\partial T_{z,u}}{\partial u} = \frac{4z}{(2-u+u\sqrt{1-4z^2})^2} \text{ and } \frac{\partial T_{z,u}}{\partial u} \big|_{u=1} = \frac{4z}{(1+\sqrt{1-4z^2})^2}.$$

To reach that $[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1} = 2 \frac{\binom{n+1}{n/2+1/2}}{n+3}$, we deal with Lagrange inversion theorem [8] p. 732. More precisely, from the functional equation $g = z + 2z^2g + z^3g^2$ followed by $\frac{\partial T_{z,u}}{\partial u} \big|_{u=1}$, putting $G = zg$, we get $G = z^2(1+G)^2$. So, putting $Z = z^2$, we get $G = Z(1+G)^2$, and we can directly apply Lagrange inversion theorem to yield $[Z^n]G = \frac{1}{n} \binom{2n}{n-1}$. The result on $[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1}$ easily ensues.

Moreover, $t_n = [z^n]T(z)$ is the number of binary trees that corresponds to Catalan numbers $\frac{2 \binom{n-1}{n/2-1/2}}{n+1}$.

So, we first get that $M_n = \frac{4n}{n+3}$.

Independently, from the exact expression of the coefficient, we also reach an exact formula for the distribution of the random variables L_n corresponding to the lifetime of the first thread in the process that return a random binary tree of size n . Indeed, we have

$$\mathbb{P}(L_n = k) = \frac{t_{n,k}}{t_n} = \frac{(k-1)(n+1)\sqrt{\pi}\Gamma(n-k)}{2^n\Gamma(n/2)\Gamma(\frac{n+3}{2}-k)}.$$

Finally, using standard probabilistic approach, we obtain the limiting distribution of the L_n :

Theorem 4. *Let L_n be the random variable corresponding to the lifetime of the first thread in the process that return a random binary tree of size n with threshold 1. Then, $\mathbb{E}(L_n) = \frac{4n}{n+3}$. Moreover the distribution L_n converges in distribution to the random variable X having $\frac{u^2}{(u-2)^2}$ as probability generating function.*

Proof: Consider the characteristic function $\phi_n(t) = \mathbb{E}(e^{itL_n})$, using classical Flajolet-Odlysko transfer theorems [7], we obtain that

$$\phi_n(t) = \frac{e^{2it}}{(e^{it}-2)^2} - 12 \frac{(e^{it}-1)e^{2it}}{n(e^{it}-2)^4} + O(n^{-3/2}).$$

So, for every $t \in \mathbb{R}$, $\phi_n(t)$ converges pointwise to $\phi(t) = \frac{e^{2it}}{(e^{it}-2)^2}$, by Lévy's continuity theorem, this implies that L_n converges in distribution to the random variable X having $\frac{u^2}{(u-2)^2}$ as probability generating function. ■

2) *Threshold 2.:* We are going to mark the nodes which are treated by the first thread. Using standard approach by

¹ $[z^n]f(z)$ classically designs the coefficient of z^n in the series $f(z)$

symbolic methods, we get the following specification for the marked class of tree:

$$\mathcal{T}^u = \mathcal{U}\mathcal{Z} + \mathcal{U}^3\mathcal{Z}^3 + \mathcal{Z}^2\mathcal{U}^2\mathcal{T}_{>1}^u + \mathcal{Z}\mathcal{U}\mathcal{T}_{>1}^u\mathcal{Z}\mathcal{U} + \mathcal{Z}\mathcal{U}\mathcal{T}_{>1}^u\mathcal{U}\mathcal{T}_{>1}.$$

Indeed, A marked tree in \mathcal{T}^u can be recursively build as follows: if its size is 1 or 3, in this case, all the node are treated by the first thread, this corresponds to the 2 first terms in the specification. Otherwise, we have 3 possible cases, the root of the tree has 2 sons L and R , and $|L| = 1$, $|R| > 1$ or $|L| > 1$, $|R| = 1$ or $|L| > 1$, $|R| > 1$ (where $|R|$ designs the size). In the both first cases, we mark the root, the singleton subtree. The remainder subtree belongs to \mathcal{T}^u . In the last case, we mark the root, the left subtree L is in \mathcal{T}^u , and the right subtree R is unmarked expect its root.

Now, directly from the specification, we get that $T_{z,u} = T(z, u)$ (and $T_z = T(z)$) is given by the functional equation:

$$T_{z,u} = uz + u^3z^3 + 2z^2u^2(T_{z,u} - uz) + zu^2(T_{z,u} - uz)(T_z - z).$$

We directly deduce that $T_{z,u} = \frac{uz(2-u^2+u^2\sqrt{1-4z^2})}{2-2z^2u^2-u^2+u^2\sqrt{1-4z^2}}$. We then can easily derive that $t_{n,k} = \sum_{j=0}^{\frac{k-3}{2}} \frac{j \binom{\frac{k-3}{2}}{j} \binom{n-k+j}{\frac{n-k}{2}}}{n-k+j}$. Again, using Odlysko-Flajolet transfer theorems, from the fact that $[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1} = [z^n] \frac{2z(1+z^2+\sqrt{1-4z^2}-z^2\sqrt{1-4z^2})}{(1-2z^2+\sqrt{1-4z^2})^2}$, we reach that

$$[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1} = 17/2 \frac{\sqrt{2}(1-e^{i\pi n})2^n}{n^{3/2}\sqrt{\pi}} + O(n^{-5/2}).$$

In fact, with a more technical study, we can show that $M_n =$

$$\frac{[z^n] \frac{\partial T_{z,u}}{\partial u} \big|_{u=1}}{[z^n] T_{z,u}} = \frac{17n^2 - 8n + 15}{n^2 + 8n + 15}$$

Theorem 5. *Let L_n be the random variable corresponding to the lifetime of the first thread in the process that return a random binary tree of size n with threshold 2. Then, $\mathbb{E}(L_n) = \frac{17n^2 - 8n + 15}{n^2 + 8n + 15}$. Moreover the distribution L_n converges in distribution to the random variable X having $\frac{u^5}{(3u^2-4)^2}$ as probability generating function.*

Proof: Again, considering the characteristic function $\phi_n(t) = \mathbb{E}(e^{itL_n})$, and using classical Flajolet-Odlysko transfer theorems, we obtain that

$$\phi_n(t) = \frac{e^{5it}}{(3e^{2it}-4)^2} + \frac{24(e^{4it}-5e^{2it}+4)e^{5it}}{n(3e^{2it}-4)^4} + O(n^{-2}).$$

So, for every $t \in \mathbb{R}$, $\phi_n(t)$ converges pointwise to $\phi(t) = \frac{e^{5it}}{(3e^{2it}-4)^2}$, by Lévy's continuity theorem, this implies that L_n converges in distribution to the random variable X having $\frac{u^5}{(3u^2-4)^2}$ as probability generating function. ■

Note to conclude this section that for every threshold the expected lifetime of the first thread is always finite (in a sense where it admits a finite limit when the size tends to the infinity). This is based on the universal shape in $(1-z/\rho)^{1/2}$ of the dominant singularity of the generating function of the

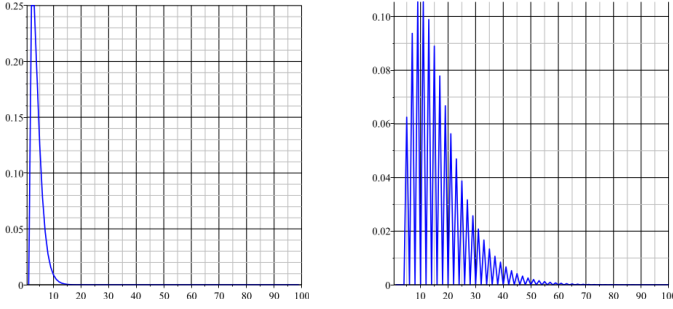


Fig. 1: Limiting distribution of lifetime with threshold 1 and 2

mean lifetime. Nevertheless, due to the increasing complexity of the specifications, the calculation is more and more tricky and becomes humanly intractable for threshold greater than 8. In order to give an intuition of the next step, we propose without explanation the specification for threshold 4:

$$\mathcal{T}^u = \mathcal{Z}^7 \mathcal{U}^7 (\mathcal{T}^4 R_4 + 4R_2 + 6R_4 + 4R_6) + \mathcal{Z}^5 \mathcal{U}^5 (4R_2 + 2R_4) + \mathcal{X}$$

where

$$\mathcal{X} = \mathcal{Z}^7 \mathcal{U}^7 + 2\mathcal{Z}^5 \mathcal{U}^5 + \mathcal{Z}^3 \mathcal{U}^3 + \mathcal{Z} \mathcal{U}$$

$$R_2 = \mathcal{Z}^2 \mathcal{U}^2 (1 + 2R_2 + R_4)$$

$$R_4 = \mathcal{Z}^4 \mathcal{U}^4 (\mathcal{T}^4 R_4 + 4R_2 + 6R_4 + 4R_6 + 1)$$

and

$$\begin{aligned} R_6 = & R_4 \mathcal{T}^8 \mathcal{U}^2 \mathcal{Z}^6 + 2R_4 \mathcal{T}^6 \mathcal{U}^4 \mathcal{Z}^6 + \\ & 5R_4 \mathcal{T}^4 \mathcal{U}^6 \mathcal{Z}^6 + 4R_4 \mathcal{T}^6 \mathcal{U}^2 \mathcal{Z}^6 + 4R_4 \mathcal{T}^4 \mathcal{U}^4 \mathcal{Z}^6 + \\ & 6R_2 \mathcal{U}^6 \mathcal{Z}^6 + 6R_4 \mathcal{T}^4 \mathcal{U}^2 \mathcal{Z}^6 + 2R_4 \mathcal{T}^2 \mathcal{U}^4 \mathcal{Z}^6 + \\ & 14R_4 \mathcal{U}^6 \mathcal{Z}^6 + 14R_6 \mathcal{U}^6 \mathcal{Z}^6 + \mathcal{U}^6 \mathcal{Z}^6 + 4R_4 \mathcal{T}^2 \mathcal{U}^2 \mathcal{Z}^6 + R_4 \mathcal{U}^2 \mathcal{Z}^6 \end{aligned}$$

By the same approach than for threshold 1 and 2, we then prove that

Theorem 6. Let L_n be the random variable corresponding to the lifetime of the first thread in the process that return a random binary tree of size n with threshold 4. Then, the random variable L_n converges in distribution to the random variable X having $-\frac{u^{11}}{(u^4-16u^2+16)(u^6-18u^4+48u^2-32)}$ as probability generating function. In particular, the mean lifetime is asymptotically 69.

IV. IMPLEMENTATION DETAILS

We implemented this parallel algorithm using the task-oriented Intel Cilk framework[11]. This model is particularly well suited for recursive processes such as the Galton Watson process. Therefore, we implemented the program in C++.

A. Memory management

One of the key issues to ensure efficiency is to avoid different threads to access the same memory zone. Mutual exclusion is necessary to avoid race conditions and false sharing. Race conditions happen when multiple threads are accessing the same memory zone, with at least one of them in write mode. They can be solved by using locks and mutexes, which are expensive in terms of computational cost. False sharing happens when several threads are trying to access areas of memory that distinct but located on the same cache line. In this case, the operating system sequentializes the memory accesses, harming the parallel performance. False sharing can

be avoided by using padding in order to make sure that variables that are accessed by different threads are far enough from each other.

In our implementation, most of the times nodes of the tree are only accessed by the thread that created them. A node contains the address of its two children. The children of a node are necessarily created by the same thread and therefore are adjacent in memory. We pre-allocate a memory block to each thread corresponding to future nodes. Once the block of a thread is filled, we create a new block and append it to the previous one, in the manner of an linked list. As a consequence, each thread has its own area of memory which is seldom accessed by other threads. The only times a node can be accessed by another thread is when a linear data structure of nodes is passed from a thread t_1 to a thread t_2 . In this case, thread t_2 will access nodes in the linear data structures which are stored in thread t_1 memory space. Though, this does not happen often and t_1 do not access those nodes. Therefore, there is few concurrency on memory accesses, and the implementation of these memory blocks makes sure that nodes located in different memory blocks are far enough from each other in memory not to be stored on the same cache line. A program can generate several trees in a row reusing the memory allocated for the previous ones.

B. Memory management

As stated in section IV-A, we implemented separate memory blocks for each thread. Each thread allocates its own memory blocks and accesses it. Moreover, to reduce the number of memory allocations (which are expensive system calls), we used a mass allocation strategy. Space for a certain number of nodes is allocated at once as a table and nodes from this table are used when necessary. The global tree is represented across these blocks by pointers between nodes. When a node is created on a thread T_i called from another thread T_j , the parent node on T_j stores a pointer to the node on T_i ; as a consequence, no thread performs any data access on another thread's memory blocks.

C. Random Number Generation

In order to obtain random bits, we had to find a pseudorandom number generator which would: noitemsep

- 1) be as fast as possible,
- 2) waste as little random bit as possible.
- 3) have a huge period,
- 4) be thread-safe,

The naive way to draw a random bit in C/C++ would be to use the `rand() % 2` instruction. Though is technique is easy to implement, it is completely inefficient: the `rand` function is rather slow (one of its step is a huge multiplication), it is neither reentrant nor thread safe, its period is 2^{32} (which is insufficient when generating several huge objects) and finally, this technique waste 31 random bits since only the last one is kept.

a) *Properties 3 and 4*: Since the classical `rand` function is not adapted, we used a pseudo random number generator from the `Boost C++` libraries: the Mersenne Twister. We used this method to generate pseudo uniform 32 bits integers. Its period is 2^{19937} and using a different generator in each thread is sufficient to guarantee it is thread-safe.

b) *Properties 1 and 2*: In order to avoid wasting random bits and accelerate the computation, we used the pseudo random integer generator to obtain a buffer of random bits. Therefore, there is no wasted random bits, except the ones that were left unused in the buffer at the end of the program, which is negligible. Since the random integer function is called 32 times less, this considerably accelerates the computation. Preserving a buffer from a call of a thread to another requires it to be stored as a global variable. When the program starts, a tabular of buffers is generated. The number of buffers is equal to the number of threads. Since this is a tabular which will be accessed by all the thread, we used padding to avoid false sharing between the random number generators. Therefore, the size of the buffers is related to the size of a cache line.

REFERENCES

- [1] Axel Bacher, Olivier Bodini, and Alice Jacquot. Efficient random sampling of binary and unary-binary trees via holonomic equations. *Arxiv*, abs/1401.1140, 2014.
- [2] Olivier Bodini, Jérémie Lumbroso, and Nicolas Rolin. Analytic samplers and the combinatorial rejection method. In Robert Sedgewick and Mark Daniel Ward, editors, *Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics, ANALCO 2015, San Diego, CA, USA, January 4, 2015*, pages 40–50. SIAM, 2015.
- [3] Stéphane Bressan, Alfredo Cuzzocrea, Panagiotis Karras, Xuesong Lu, and Sadegh Heyrani Nobari. An Effective and Efficient Parallel Approach for Random Graph Generation over GPUs. *J. Parallel Distrib. Comput.*, 73(3):303–316, March 2013.
- [4] N. G. de Bruijn, D. E. Knuth, and S. O. Rice. The average height of planted plane trees. In R. C. Read, editor, *Graph Theory and Computing*, pages 15–22. Academic Press, 1972.
- [5] Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004.
- [6] Philippe Flajolet, Zhicheng Gao, Andrew M. Odlyzko, and L. Bruce Richmond. The distribution of heights of binary trees and other simple trees. *Combinatorics, Probability & Computing*, 2:145–156, 1993.
- [7] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. *SIAM J. Discrete Math.*, 3(2):216–240, 1990.
- [8] Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.
- [9] Philippe Marchal. Generating random alternating permutations in time $n \log n$. working paper or preprint, December 2012.
- [10] Jean-Luc Remy. Un procédé itératif de dénombrement d’arbres binaires et son application a leur génération aléatoire. *ITA*, 19(2):179–195, 1985.
- [11] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, November 2001.